stability, reduced sensitivity to missing-value artifacts and improved cross-domain transferability, validating the effectiveness of hybrid multi-source architectures for next-generation decision support systems.

Experimental validation was conducted using benchmark datasets:

1. UCI Multisource Activity and Process Analytics Repository for heterogeneous event-sequence modelling,
2. Industrial Sensor-Text Fusion Corpus for cross-modal diagnostics and classification,
3. Smart-Operations Log Intelligence Dataset for decision-support scenario evaluation,
4. Synthetic Multi-Modal Stress-Test Suite for robustness and distribution-shift analysis.

The system achieved consistent performance improvements under partial-data and noise-rich scenarios, demonstrating scalability and practical applicability in real-world intelligent analytics platforms.

References

1. Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press.
2. Vaswani, A., et al. (2017). Attention Is All You Need. NeurIPS.
3. Xu, C., Tao, D., Xu, C. (2013). A Survey on Multi-View Learning. Neural Computing and Applications.
4. Ribeiro, M. T., Singh, S., Guestrin, C. (2016). Why Should I Trust You? Explaining the Predictions of Any Classifier. KDD.
5. Zhang, D., et al. (2022). Hybrid Deep Architectures for Multi-Source Analytics in Decision Support Systems. Expert Systems with Applications.

DOI 10.70286/ISU-14.01.2026.005

THE TEST PYRAMID IN SOFTWARE ENGINEERING: WHY AN OVERRELIANCE ON INTEGRATION TESTS DOES NOT GUARANTEE QUALITY

Kazantseva Sofiia
Student
Software Engineering Department
Kharkiv National University of Radio Electronics, Ukraine

Software quality is often (incorrectly) “measured” by the sheer number of tests especially integration tests. In practice, this leads to a misleading conclusion: if there are many integration tests, the system must be reliable. However, in quality engineering it is not only the quantity that matters, but which tests you have, where they sit in the hierarchy, what information they provide when they fail, and how costly they are to maintain. The test pyramid concept emerged as a response to these concerns: it does

not reject integration tests, but emphasizes that an excessive share of them often reduces the effectiveness of a testing strategy.

The test pyramid describes a balanced distribution of tests by levels: at the base are numerous, fast unit tests; above them are fewer integration and component tests; at the top are end-to-end (E2E) tests that emulate real user behavior. The idea is straightforward: the closer a test is to the "real system" (network, database, external services, UI), the more expensive, slower, less stable, and less diagnostic it tends to be [1]. This is why "many integration tests" is not a quality indicator it is often an indicator of high verification cost and poor fault localization.

The first reason is the speed of feedback. Unit tests execute in milliseconds and give developers immediate signals that a specific function or class has broken expected behavior. Integration tests especially those that spin up real databases, message brokers, or multiple services can take minutes [2]. This increases CI pipeline duration, slows interactive development, and effectively delays defect detection. As a result, teams discover bugs later, when they are more expensive to fix, and release risk increases. In other words, what grows is not quality, but the cost of achieving quality.

The second reason is flakiness and environmental dependence. Integration tests often depend on networks, concurrency, timing, database state, data availability, container configuration, and parallel execution [3]. They can fail not because of a code defect, but due to environmental instability or race conditions. When teams frequently see flaky failures, a dangerous effect emerges: failing tests stop being taken seriously. This undermines trust in CI as a "source of truth" and, paradoxically, reduces real quality, even if the number of tests is high.

The third reason is low diagnostic value. A good test should not only detect a failure, but also help locate its cause quickly. Unit tests typically point to an exact function/method and a specific scenario. Integration tests often fail "somewhere in between" components: the issue might be in ORM mapping, database schema, serialization configuration, an API contract, or even test data. Developers spend significantly more time localizing the problem [4]. If there are hundreds of such tests and they are poorly isolated, even a small change can trigger a cascade of failures. Thus, the number of integration tests may correlate not with quality, but with time spent investigating.

The fourth reason is excessive coupling and inhibited evolution. Integration tests frequently validate a specific implementation rather than behavioral invariants. As a result, they can "cement" the system's structure: refactoring, schema changes, or interface optimizations often require widespread test rewrites [5]. Teams then start avoiding improvements because they fear breaking the test suite. Architectural quality degrades, and tests become a barrier rather than an enabler of change.

It is important to stress that integration tests are necessary. They reveal classes of defects that unit tests cannot catch: incorrect transactions, migrations, configuration issues, faulty SQL, serialization/deserialization problems, and contract mismatches between services. The problem arises when integration tests are expected to cover everything business logic, validation, edge cases, and invariants [6]. This is a strategic mistake because such properties are best tested closer to the code: faster, more precisely, and more deterministically.

A practical approach is a rational distribution of responsibility across levels. The core principles can be stated as follows:

1. Business logic and edge cases should be covered primarily by unit/component tests, where dependencies are controlled (mocks/fakes/stubs).

2. Integration tests should focus on the system's "seams": the database, messaging, external APIs, serialization, security configuration, and transactional behavior. They should be targeted, not all-encompassing.

3. E2E/UI tests should be kept to the minimum required for critical user journeys (for example, 5–20 key scenarios), because their cost is the highest.

It is also worth mentioning tools and techniques that reduce integration-testing pain without losing its benefits. For example, containerized dependencies (such as running a database in a container) improve reproducibility, but they do not eliminate the inherent cost of integration tests [7]. A strong alternative to excessive end-to-end integration is contract testing (validating service interfaces), as well as component testing (testing a service almost like a black box, but with controlled dependencies). Where possible, hermetic tests tests maximally isolated from external factors help prevent flakiness.

In conclusion, a high-quality testing strategy is not "the more, the better," but an optimization of three dimensions: accuracy (what exactly we validate), speed (how quickly we get signals), and maintenance cost (how much time is spent fixing tests and investigating failures). A large number of integration tests can create an illusion of reliability, while actually increasing CI instability, slowing development, and reducing the team's ability to evolve the system. That is why the test pyramid remains a practical engineering principle: quality comes from a balanced hierarchy, not from dominance of the most expensive level of verification.

References

1. Cohn, M. Succeeding with Agile: Software Development Using Scrum. Upper Saddle River, NJ : Addison-Wesley Professional, 2009. 504 p.
2. Fowler, M. Test Pyramid [Electronic resource]. 2012. Available at: https://martinfowler.com/bliki/TestPyramid.html (Accessed: 13 Jan 2026).
3. Fowler, M. The Practical Test Pyramid [Electronic resource]. 2018. Available at: https://martinfowler.com/articles/practical-test-pyramid.html (Accessed: 13 Jan 2026).
4. Fowler, M. On the Diverse and Fantastical Shapes of Testing [Electronic resource]. 2021. Available at: https://martinfowler.com/articles/2021-test-shapes.html (Accessed: 13 Jan 2026).
5. Google Testing Blog. Just Say No to More End-to-End Tests [Electronic resource]. 2015. Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html (Accessed: 13 Jan 2026).
6. Google Testing Blog. SMURF: Beyond the Test Pyramid [Electronic resource]. 2024. Available at: https://testing.googleblog.com/2024/10/smurf-beyond-test-pyramid.html (Accessed: 13 Jan 2026).
7. Meszaros, G. xUnit Test Patterns: Refactoring Test Code. Boston : Addison-Wesley, 2007. 883 p.